MLP Coursework 1: Learning Algorithms and Regularization

s1886585

Abstract

This coursework aims to show the use of different Stochastic Gradient Descent based algorithms to perform classification in the extended MNIST data set, using a 3 hidden-layer neural network. Firstly, experiments using RMSprop and ADAM learning rules are performed. Next, a cosine annealing learning rate scheduler is implemented and tested, using Stochastic Gradient Descent and ADAM learning rules. Finally, regularization is added to the neural network, by two different approaches: weight-decay and L2 regularization using ADAM and the scheduler. All the implementation hyperparameters are chosen by maximizing the error on a validation set. The implementations are then compared using a test set.

1. Introduction

During the last years, several techniques to improve the performance of Neural Networks (NNs) have been developed. Some of this techniques focus on refining the underlying optimization process, by modifying the way the NN parameters are updated. Different adaptations of the *Stochastic Gradient Descent* (SGD) algorithms with varying learning rate, weighted gradient or regularization have been proposed (Ilya Sutskever & Hinton, 2013), (John Duchi, 2011). The aim of this coursework is to explore and compare the performance of different adaptations. Regarding the learning rule, recently proposed *RMSprop* (Tieleman & Hinton) and *ADAM* (Kingma & Lei Ba, 2017) are implemented. Furthermore, a Cosine annealing learning rate scheduler is also implemented (Loshchilov & Hutter, 2018). Finally, experiments regarding the use of regularization with the scheduler are performed.

In order to prove the use of this algorithms, the extendet MNIST (EMNIST) balanced data set is used (Gregory Cohen & van Schaik, 2017). The objective is to increase as much as possible the accuracy of classification of 47 different classes, from the data set. Each sample is an 28x28 pixel image, representing a handwritten character, which is transformed into a 784 vector, which then is propagated through the NN, and classified in one of 47 classes. Cross entropy softmax error was used to optimize the NN and to compare results between implementations.

2. Baseline systems

To begin, experiments involving a NN with different architectures, where trained by using the SGD method. SGD is a type of gradient descent method in which the parameters are updated by batches, selected randomly. The method consists in updating the parameters of the NN by the following rule,

$$w^{(t)} = w^{(t-1)} - \eta g^{(t-1)}, \tag{1}$$

where η is the learning rate of the NN and g is the gradient of the loss function w.r.t to a given parameter w. This will make the parameters move towards the direction where the loss is decreasing most rapidly. This operation is applied to a whole mini-batch at the same time. A correct choose of η is important for the training of the neural networks, since very small numbers will make convergence vary slow or even impossible, while very large value will lead to non-convergence. Therefore η becomes a hyperparameter which needs to be tuned. Since the loss function is non-convex, there is no guarantee of convergence to a global minimum, and also convergence to a local minimum can take long times or lead to overfitting, so the total number of epochs for which the algorithm will run needs to be tuned to. Finally, Different sizes of minibatches will change the time performance of the NN training, so a third hyperparameter arises. Tuning of these hyperparameters on a two layer with 100 relu per layer architecture is described. Once the hyperparameters are tuned, architectures with three, four and five layers, each with 100 relu units are trained, and compared using test error.

2.1. Early stopping implementation

The first hyperparameter to be tuned in order to optimize the time to train the NN is the number of epochs. However, choosing this parameter without setting the learning rate and batch size could be problematic since the amount of epochs needed to reach convergence is highly dependent in these two other hyperparameters. For this reason, an early stopping algorithm was implemented. By doing so, not only time resources are managed more efficienlty but also overfitting is prevented. This algorithm was implemented by modifying the `train` method in the `Optimizer` class. The implementation consists in adding a `break` statement to the `for loop` if after certain amount of (N) epochs, the validation error is not improving, and returning the model with the best validation error. Although a new hyperparameter appears, it can be tuned manually without high risk of loosing performance. Values between $N = 5$ and $N = 10$ show good performance, and reduced drastically the computation times.

2.2. Batch size

Secondly, experiments on the performance of different batch sizes were made. Tuning the batch size aims to improve

the time performance rather than the generalization performance (Bengio, 2012), (Nitish Shirish Keskar & Tang, 2017). In order to find the best batch size, this hyperparameter was set to values from 10 to 100, and the amount of epochs, and time for each epoch necessary to achieve a 0.8 validation error was measured. Once the validation error was lower than 0.8 the optimizer `for loop` was stopped. This 0.8 value was selected manually because it is roughly the error where the validation error and the train errors started to diverge. The results for this experiment are shown in figure 1.

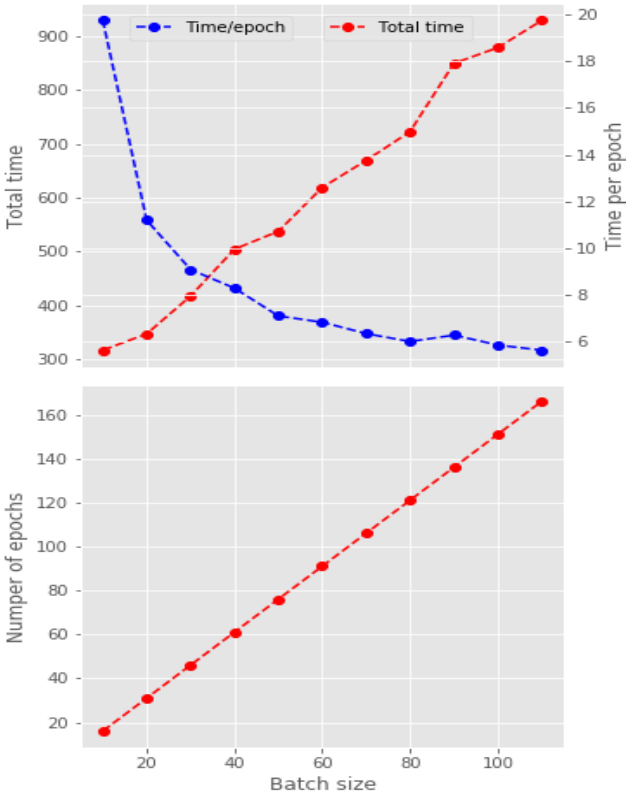

Figure 1. Time performance for different batch sizes. The upper plot shows the total time and the time by epoch in seconds. The lower one shows the number of epochs required to achieve a 0.8 validation error for each batch size.

There is a clear trade-off between the number of epochs required to reach a validation error and the time per epoch. This can be explained by the fact that bigger batches sizes take advantage of vectorization, but since the amount of minibatches decreases, the step size also decreases. When using small batch sizes, more iterations are required per epoch, but a bigger step is performed, thus decreasing the number of epochs to reach certain error. A batch size of 10 was used for the first experiments, since it considerably decreased the training time, without high loss in performance.

2.3. Learning rate

Finally, the learning rate (η) which minimized the validation error was found. Initially, A broad tuning of η was done manually by changing its value from 1e-6 to 0.1, in big steps. If η was in the smallest values, convergence took long times, however for the values around 0.1 no convergence was found. Then a more narrow tuning was done by looking in values around 0.01. Results of this tuning are depicted in figure 2. By increasing the learning rate, there is a tendency to also decrease the training time, since bigger steps are taken towards a minimum. However, very big steps will make it more improvable to reach a minimum. A value of $\eta = 0.013$ displayed the best validation error. NNs with three, four and five Relu layers were trained with these hyperparameters. The results of the test error for these were: $E_2 = 0.545$, $E_3 = 0.582$, $E_4 = 0.551$ and$E_5 = 0.538$ respectively.

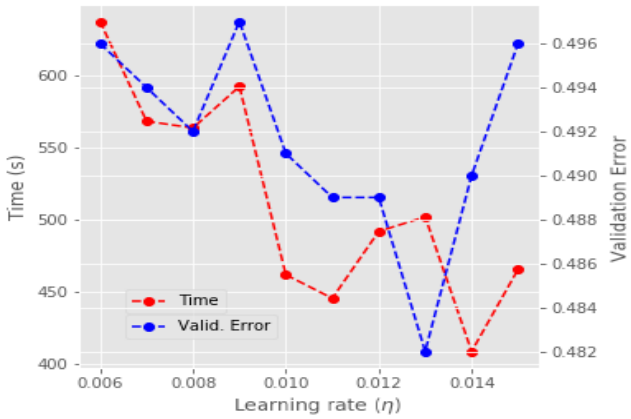

Figure 2. Performance of the NN when changing the learning rate. The total training time and errors are displayed.

3. Learning algorithms – RMSProp and Adam

Loss functions involved in NNs usually have the particularity that the negative gradient of the loss function w.r.t to the parameters at some point do not necessarily points towards the a minimum. In this sense, moving opposite to the gradient does not necessarily implies efficient steps towards a minimum of the loss function. For this reason, different gradient descent like algorithms in which the quantity by which the gradient is weighted in order to update the parameters changes from epoch to epoch have been proposed. Initially two of these learning algorithms will be implemented and discussed: RMSprop and ADAM.

3.1. RMSProp

The general idea behind RMSprop is to give a greater weight to the update of parameters which make the function go towards a minimum fast, and vice versa. This can be done by introducing the second moment, which for iteration t and a given parameter is,

$$m_2^{(t)} = \beta m_2^{(t-1)} + (1-\beta)g^2, \tag{2}$$

where β is a new hyper-parameter that weights the average between g^2 and the previous value of the moment, and g is the gradient with respect to the given parameter. The parameter is then updated by the following rule,

$$w^{(t)} = w^{(t-1)} - \frac{\eta g}{\sqrt{m_2^{(t)} + \epsilon}}, \tag{3}$$

where η is the learning rate and ϵ 10^{-8} is used to prevent division by zero.

Two hyper-parameters need to be tuned when using the RMS-prop algorithm: learning rate (η) and the second moment parameter (β). Both η and β are initially manually tuned by trying to minimize the validation error. Then, a fine scan is done to η. Finally, once an optimal η is selected, a fine scan is performed on β. Results of this process is displayed in figures 3 and 4. To maximize the validation error, the hyperparameters were choosen to be $\eta = 4 \times 10^{-4}$ and $\beta = 0.998$. The error on the test set for this configuration was 0.561.

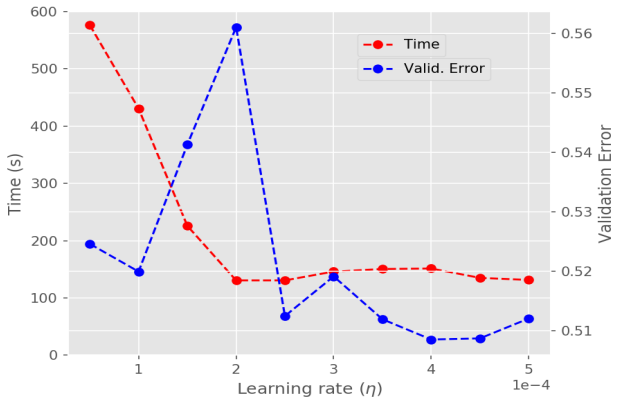

Figure 3. Validation error and training time for different learning rates in the 10^{-4} order of magnitude, using RMSprop

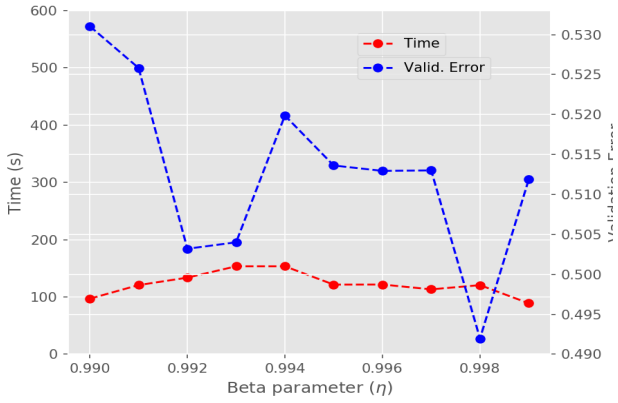

Figure 4. Validation error and training time for different β between 0.99 and 0.999 for RMSprop.

3.2. ADAM

Similar to RMSprop, the ADAM learning algorithm, penalizes the parameters with a high second moment of the gradient. However, the first moment of the gradient is also used, favouring the weights which move towards a minimum. First moment is therefore defined similarly to the second moments from RMSprop as follows,

$$m_1^{(t)} = \beta_1 m_1^{(t-1)} + (1 - \beta_1) g. \tag{4}$$

Here, β_1 denotes the weighting for the average between the previous first moment and the gradient. The second moment is the same as in RMSprop (however β is now notated as β_2). Also, two new quantities are introduced: The biased-corrected first and second moment estimations which are, $\hat{m}_i^{(t)} = m_i^{(t)}/(1-\beta_i{}^t)$, with $i \in 1, 2$, and t being the step count. Biased-corrected moments are useful since the moments are initialized in zero and β_1, β_2 are close to one, the first iterations will be biased towards zero. Notice that $\hat{m}_i \to m_i$ as $t \to \infty$. The parameters update is then,

$$w^{(t)} = w^{(t-1)} - \frac{\eta \hat{m}_1^{(t)}}{\sqrt{\hat{m}_2^{(t)} + \epsilon}}, \tag{5}$$

As in the previous algorithms, the learning rate was broadly tuned manually. Then, a scan was performed around the learning rate that manually minimized the validation error and the default value of $\beta_1 = 0.9$. The results of the scanning is depicted in figure 5.

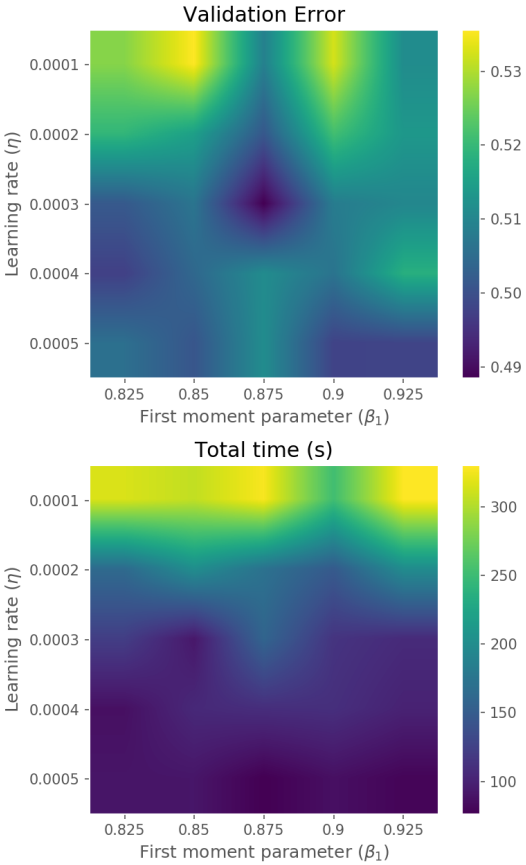

Figure 5. Validation error an training time for different values of learning rate and β_1 using ADAM.

Once η and β_1 are tuned, β_2 is optimized by performing a scan between $\beta_2 = 0.9985$ and $\beta_2 = 0.9995$. The hyper-parameters which lead to the least validation error were: $\eta = 3 \times 10^{-4}$, $\beta_1 = 0.875$ and $\beta_2 = 0.9992$. The error on the test set with these configuration was 0.553.

3.3. RMSProp, Adam and SGR

As mentioned, the intention behind using RMSprop and ADAM is to make steps which will converge faster towards a minimum. It is possible to notice if figure 3 and 5 how a proper choose of learning rate can decrease the training time by a factor of 5 (from ~500s to ~100s), without affecting negatively (or even affecting in a positive way) the validation error, when compared to SGD. Also, the validation error is minimized in the area where the training time reaches a flat surface, both for ADAM and RMS prop, since this is the optimal point of the trade-off between choosing a small learning rates which takes very long times to converge, and high learning rates which can lead to non convergence. Finally, time is relatively invariant to β_2 and β_1 for a proper choice of η.

4. Cosine annealing learning rate scheduler

Setting a correct value for the learning rate can lead to a successful implementation of a NN. However this is not a trivial task. One idea to facilitate this process is to use a learning rate which varies from epoch to epoch. The cosine annealing learning rate scheduler is an algorithm used for this task. In this algorithm, the learning rate starts at a maximum value (η_{max}) and decays by a cosine function until a value (η_{min}). The scheduler has the option to be restarted, so it would take a fraction of the initial η_{max} value and decay again by the cosine function. The learning rate is then,

$$\eta_t = \eta_{min}^{(t)} + 0.5(\eta_{max}^{(t)} - \eta_{min}^{(t)})(1 + \cos(T_{cur}/T_i)). \quad (6)$$

Since the implementation of the scheduler wanted to be completely tested, the early stopping function was partially blocked. This will allow the training process to run until the end, but will store the model with the best validation error, to avoid overfitting. In order to allow the implementation of the scheduler to continue from any epoch, the following mathematical procedure was used. According to (Loshchilov & Hutter, 2018) the scheduler can restart every T_i, allowing it to take high values again. Every time it is restarted, the maximum learning rate ($\eta_{max}^{(i)}$) and T_i, are multiplied by a factor. In this sense, after i restarts, $T_i = T_0 \times T_{mult}^i$ where T_0 denotes the first size of epochs for restart. Equally, $\eta_{max}^{(i)} = \eta_{max}^0 \eta_r^i$. For convergence the hyperparameters denoting the geometrical rate should be $T_{mult} \geq 1$ and $\eta_r < 1$. Now, assuming the epoch number ϕ is given, T_i and η_{max}^i can be found if i is known. Therefore the task is to find i given ϕ. In general, ϕ can be written as the sum of all previous T_i plus the number of epochs in the current T_i denoted α. Then,

$$\phi = \sum_{i=0}^{k-1} T_i + a = \sum_{i=0}^{k-1} T_0 T_{mult}^i + \alpha, \quad (7)$$

by using the geometric series formula,

$$\phi = T_0 \left(\frac{T_{mult}^k - 1}{T_{mult} - 1} \right) + \alpha, \quad (8)$$

for $T_{mult} \neq 1$. The case for $T_{mult} = 1$, will be studied at the end. Now, α is also an unknown value. However, for any value of α k is the same, so α will be temporally removed from the equation. Finally solving for k,

$$k' = \log_{T_{mult}} \left(\frac{\phi(T_{mult} - 1)}{T_0} + 1 \right) \quad (9)$$

Now, this process was ignoring α, and thus the result will be a decimal number. To take into account the existence of α, the integer part of this result is taken $k = \lfloor k' \rfloor$, finding the number of iterations so far. With this value, now all the other values such as α, $\eta_{max}^{(i)}$ and T_i can be found by replacing in the above equations, allowing to implement the scheduler from any epoch. The case where $T_{mult} = 1$ is simplified, since $\phi = k_{mult} + \alpha$, therefore, $k = \lfloor \phi / T_{mult} \rfloor$ and $\alpha = \phi \mod (T_{mult})$.

The scheduler was tested in four different scenarios: SGR and ADAM, each with no restarts and restart at $T_i = 25$, with $T_{mult} = 3$ for 100 epochs. In the SGR case, $\eta_{max} = 0.1$ and $\eta_{min} = 0.01$, since these values showed good performance in the SGD implementation in section 2. Consequently, for the ADAM learning algorithm $\eta_{max} = 4 \times 10^{-4}$ and $\eta_{max} = 2 \times 10^{-4}$. The moment parameters are for ADAM are the same as in section 3. The learning curves for the restart cases are displayed in figures 6 and 7 for SGD and ADAM respectively. Figure 6 also show the behaviour of the learning rate.

From the results it is possible to observe a very good performance on the training error. This happens since the optimization process involves the test data, and the scheduler allows the optimizer to decrease even more the training error. However, the validation error reaches a minimum and then increases, especially for the SGR case. This is a clear sign of overffiting. Implementing a regularization technique should increase the performance on the validation set. Also, in figure 7 it is possible to notice that before the restart, the errors start to converge since the steps become very small, however the restart allows the error to decrease again. The test set results for these four cases were: 1.180, 0.970, 0.662, 0.554 for SGD with no restart, SGD with restart, ADAM with no restart and ADAM with restart, respectively.

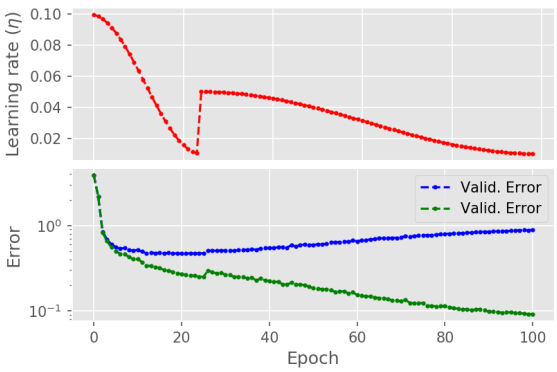

Figure 6. Learning curve of train and validation set when using a cosine scheduler with restart at $T_i = 25$ in SGD. The learning rate is also depicted.

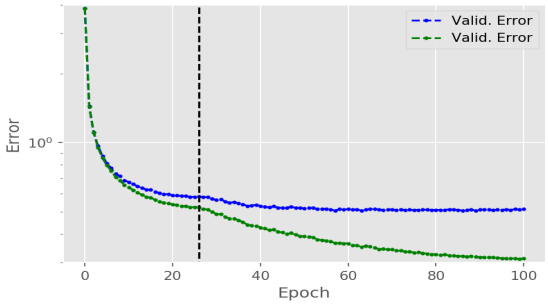

Figure 7. Learning curve of train and validation set when using a cosine scheduler with restart at $T_i = 25$ in ADAM. The black line represents the restart.

5. Weight decay and L2 regularization

The implementation of a scheduler shows a very promising optimization behaviour over the training set, however the generalization performance shows overfitting. For this reason the implementation of a regularizeris used. [REF] states that for a constant learning both L2 regularization and weight decay are the same process, but when a scheduler is implemented they can differ. Weight decay consists in ADAM consists in adding a regularization term to the update rule in the following way,

$$w^{(t)} = w^{(t-1)} - \eta \left(\frac{\hat{m}_1}{\sqrt{\hat{m}_2^{(t)} + \epsilon}} + \lambda w^{(t)} \right), \qquad (10)$$

where λ is the weight update hyperparameter (Loshchilov & Hutter, 2018). Tuning of this hyperparameter is depicted in figures 8 for a cosine scheduler with no restarts. The same process was done to the scheduler with restarts. Notice, that the regularization has the tendency to find the best validation error around the final epochs. In both cases the best validation was achieved in the epoch 100. This means that the learning process is slowed down, so using more epochs should produce better results. The λ parameters which displayed the best validation error were 0.19 and 0.15 for a scheduler with no restart and with restart respectively. The error on the test set for these values was 0.498 and 0.492.

Finally, L2 regularization was also implemented to the cosine scheduler with no restarts. L2 regularization is performed by changing the update of g to $g^{(t)} = \nabla(w) + \theta w$, where θ is the L2 penalization hyperparameter. This type of regularization introduces coupling between the θ and the other ADAM hyperparameters, making the tuning of θ more complicated. The regularization was tested using $\eta_{max} = 6 \times 10^{-3}$ and $\eta_{min} = 2 \times 10^{-3}$, with the other ADAM parameters remaining the same. The scan of θ is shown in figure 9. The best validation error obtained when $\theta = 3 \times 10^{-4}$, with a test error of 0.435.

6. Conclusions

Experiments using the EMNIST on a NN were performed, by changing it's architecture, learning rule, scheduler and regularization. Changes in the learning rule showed decreasing training times of the NN. The use of an scheduler allowed the training error to decrease to a lower value lower than the ones obtained without the use of it. However, generalization was highly affected by the use of it, due to overfitting. The use of a restart on the scheduler was succesfully tested. To overcome the probelm of overfitting when using the scheduler, weight decay and L2 regularization were implemented, leading to the highest performance of the NN. The result of all the different test sets is displayed in table 1. An improvement on the performance of the NN is displayed, however better results could be achieved by scanning over more hyperparameters, and using a bigger amount of epochs. Pre-processing the data set by adding noise can also lead to better results (Dan Claudiu Ciresan, 2010)

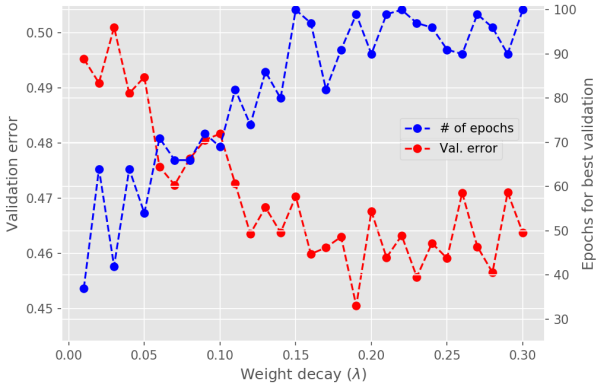

Figure 8. Best validation error and number of iterations required to obtain the best validation error, for different weight decay parameter (λ) using cosine annealing scheduler with no restarts, and ADAM learning rule.

Algorithm	Test error	Acc. (%)	Time(s)
Cos L2. No R.	0.435± 0.008	85.0	408
Cos W.D. No R.	0.489± 0.009	83.7	301
Cos W.D. R	0.492± 0.010	83.8	319
SGD 5 Layer	0.538± 0.011	82.6	357
SGD 2 Layer	0.545± 0.011	83.0	223
SGD 4 Layer	0.551± 0.012	83.1	303
ADAM	0.553± 0.011	82.3	159
Cos Sch. ADAM R	0.554± 0.011	82.8	369
RMSProp	0.561± 0.012	82.8	125
SGD 3 Layer	0.582± 0.012	82.3	277
Cos Sch. ADAM no R.	0.662± 0.014	82.5	346
Cos Sch. SGD R.	0.970± 0.025	81.9	363
Cos Sch. SGD no R.	1.180± 0.031	81.9	369

Table 1. Test error, accuracy and training time for all the different algorithms. R and no R stands for restart and no restart. Results are ordered by Test error performance.

The most difficult part of training the NN has to do with the proper selection of its hyperparameters, specially since the possible set of combinations for them scales exponentially with the number of hyperparameters. Then, it is necessary

to perform careful experiments when choosing them for baseline and non-complex systems, specially when time resource is reduced. A good way to ensure better results, is to cycle more than one time over the hyperparameters, specially on the learning rate (Bengio, 2012).

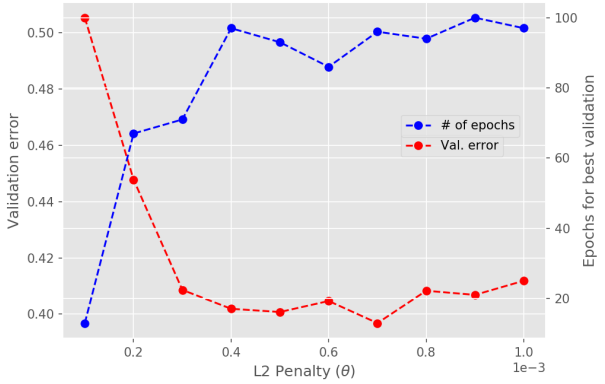

Figure 9. Best validation error and number of iterations required to obtain the best validation error, for different L2 penalties (θ) using cosine annealing scheduler with no restart, and ADAM learning rule.

References

Bengio, Joshua. Practical recommendations for gradient-based training of deep architectures, 2012. URL https://arxiv.org/abs/1206.5533.

Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella Juergen Schmidhuber. Deep big simple neural nets excel on hand-written digit recognition, 2010. URL https://arxiv.org/abs/1003.0358.

Gregory Cohen, Saeed Afshar, Jonathan Tapson and van Schaik, AndrÃl'. Emsnist: an extension of mnist to handwritten letters. *arXiv arXiv:1702.05373v2*, 2017. URL https://arxiv.org/abs/1702.05373.

Ilya Sutskever, James Martens, George Dahl and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning, 2013. URL http://proceedings.mlr.press/v28/sutskever13.pdf.

John Duchi, Elad Hazan, Yoram Singer. Adaptative subgradient methods for online learning and stochastic optimization, 2011. URL http://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

Kingma, Diederik P. and Lei Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv arXiv:1412.6980v9*, 2017. URL https://arxiv.org/abs/1412.6980.

Loshchilov, Ilya and Hutter, Frank. Fixed weight decay regularization in adam. *arXiv arXiv:1711.05101V2*, 2018. URL https://arxiv.org/abs/1711.05101.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal Mikhail Smelyanskiy and Tang, Ping Tak Peter. On large-batch training for deep learning: Generalization gap and sharp minima, 2017. URL https://arxiv.org/abs/1609.04836.

Tieleman, Tijmen and Hinton, Geoffrey E. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.